The University of Hong Kong

Department of Computer Science

COMP3362

Final Report

Whiskey Classifier

KIM, Sukmin **3035445479**

LEE, Sang-hyeon **3035396173**

List of Figures

List of Tables

Table of Contents

Objectives

Whiskey is one of the most popular and common alcoholic beverages, and it has a very long history. Therefore, there are various types of whiskey and, each whiskey has unique flavor, scent, and age even though they belong to the same type and brand of whiskey. As a result, it has been hard for people who are not familiar with whiskey to understand the whiskey by looking at the bottle, and the objective of the project is to build an application to detect and classify whiskey bottles when users provide an image of a whiskey.

Highlights

- Dataset prepared by data crawling and labelling
- AI model to classify whiskeys implemented, with in-depth optimizations and error analysis
- Web application to interact with users implemented

Tasks Achieved

Dataset Preparation

There are hundreds of whiskeys in the world, and it would be impossible to build an AI model to classify all of them within limited time and resources. Therefore, we decided to narrow down the scope of the whiskeys to 30 most popular whiskeys sold in Market Place by Jasons online store.

To train the model, we needed images of the target whiskey, but we did not have any public dataset available to serve the objectives of the project. Therefore, we had to build a web crawler to collect the images using Python and icrwaler library which provides GoogleImageCrawler class to support Google Advanced Image Search. With the advanced image search, we were able to collect images of whiskey with successfully ignoring low-resolution images. However, some of the images collected were irrelevant and inappropriate to be used as dataset. Therefore, the number of selected images were around 2,700. As the project progresses, it is found that there is not enough dataset for testing the model, therefore, around 700 more images were collected and labelled. The implementation of the data crawler was very simple and it is provided in Figure 1.

```
import os
from icrawler.builtin import GoogleImageCrawler

for name, path in whiskey_list:
    filters = dict(size='large')
    os.makedirs(path)
    print("The new directory is created! - ", path)
    google_crawler = GoogleImageCrawler(feeder_threads=2,
                                        parser_threads=2,
                                        downloader_threads=8,
                                        storage={'root_dir': path})
    google_crawler.crawl(keyword=f'{name} photography',
                         language="us",
                         max_num=50,
                         filters=filters)

    print("image saved - ", name)
```

Figure 1. Implementation of data crawler

With our data crawled from the internet, we have started manual labeling of bounding boxes with Annotate in Roboflow framework. With several available labelling software available, we have selected Roboflow because it was fast to label and edit bounding boxes and more importantly easy to export dataset into COCO format. With Roboflow, we just need to drag the mouse pointer to indicate the bounding boxed and label with the existing classes or create a new class to label the images (see Figure 2). Without first downloading large sized dataset into local machine then transferring into CS GPU Farm, we can easily execute terminal code that Roboflow provided so directly download images into the CS GPU Farm using curl command.

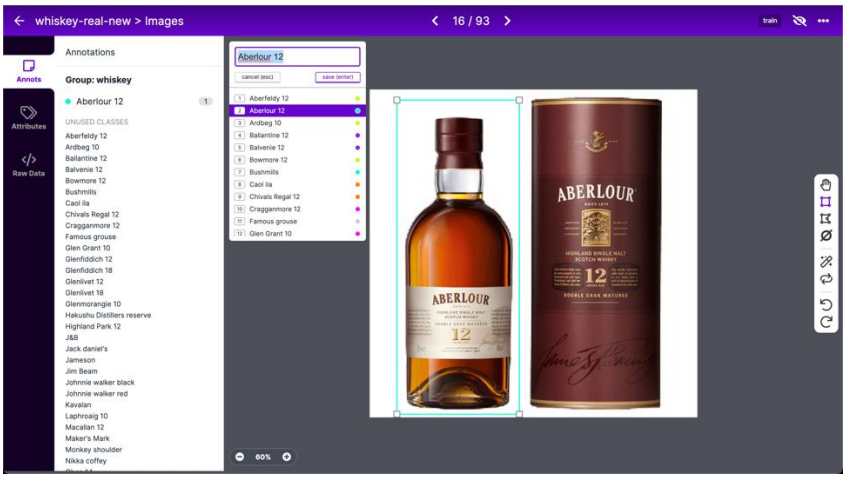


Figure 2 User interface of Roboflow for labelling the images

Data engineering

First, we started with 37 different types of whiskeys with total 2,678 images collected and labeled. However, we had initial filtering on excluding images that are redundant thus resulting images have class imbalance as shown in Figure 3. Therefore, we again removed 10 classes with <30 number of boxes, resulting in 27 classes in our final dataset. Although there still exists class imbalance between different classes (Laphroaig 10 counts 161 box annotations while Glenfiddich 12 counts only 32 box annotations), we did not make further improvement on our dataset because our focus in the project is not to create high quality data, and this well represents the dataset we usually face on practical cases.

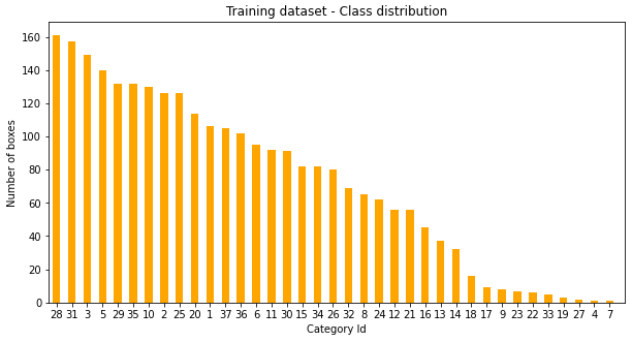


Figure 3 Class distribution of the initial training data

While crawling, we have realized that many of the images are white background image. Merely using these image

types in our training will not learn different patterns and features that we detect whiskey bottles in real life. Therefore, we have made additional effort to increase the variability of our dataset. For example, in Figure 4 with Highland Park 12 class, dataset type includes different backgrounds: white and non-white, occlusion of the bottle, image including only part of the bottle enlarged, empty bottles without liquid inside and others. Furthermore, later we try to use different image augmentation combinations on training dataset so it can learn more diverse features in the image.

Figure 4 Various data types of Highland Park 12

We have done short data exploration to validate our dataset created. First, the box size respect to image size ratio was analyzed, and the result was visualized using histogram and bar chart (see Figure 5). Large portion of the images were placed between 0.2 and 0.4 in terms of ratio and the class with largest bounding box size was Woodford Reserve and the class with smallest bounding was Glenfiddich 12.

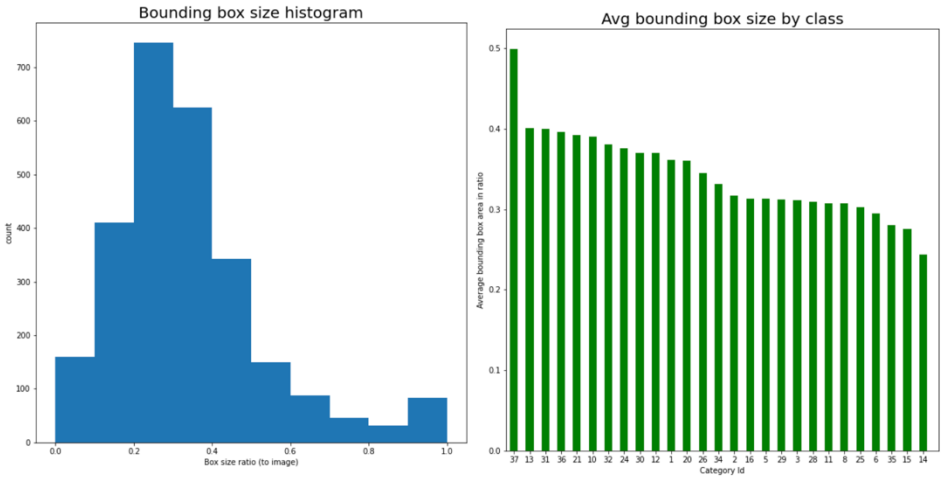


Figure 5 Data visualization on bounding box size

Then, we also looked at the aspect ratio of the box. Most of the data whiskey had ratio smaller than 0.4 because whiskey bottles in general are tall and slim. However, Woodford Reserve has a unique shape (see Figure 6) and, therefore, it has an average bounding box ratio of 0.5 which is distinct.

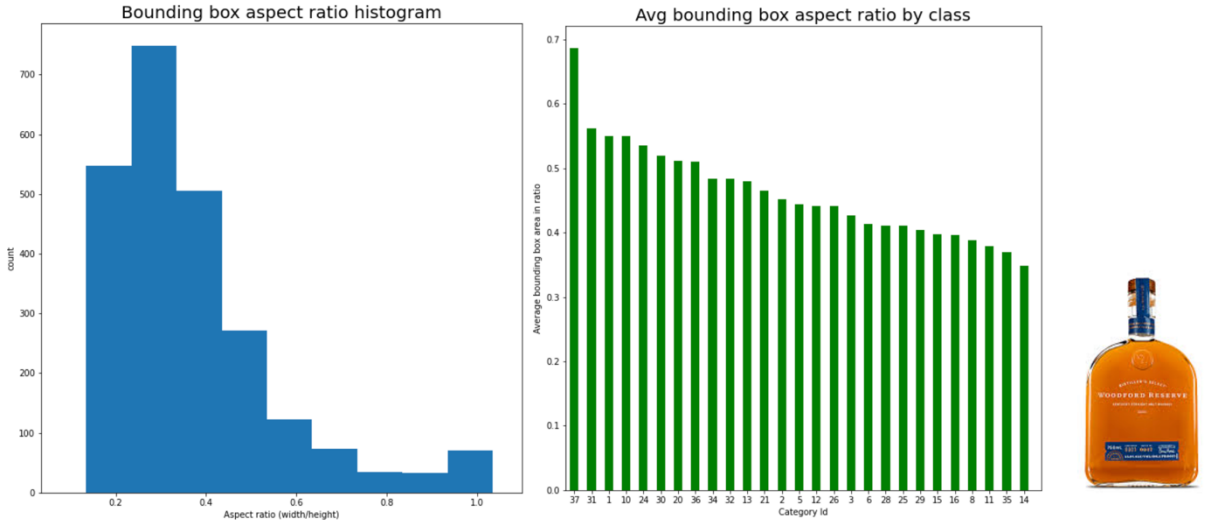


Figure 6 Data visualization on bounding box ratio and image of Woodford Reserve

Finally, we could easily split train, validation, and test datasets using Roboflow split feature. In the beginning in the project, we have divided the dataset into 8:1:1 ratio for train:validation:test, but later we have decided to increase training dataset size so merged training and test dataset and created new test dataset. We did not use test dataset until the very end of the project since test dataset roles to test performance of our model on unseen dataset.

However, in the middle of the project, we have made huge mistake on this split. We believed that Roboflow will use stratified split, which it essential to create reliable split data with similar class distribution, but it does biased splitting in default. If we have done data analysis after splitting, we could detect this earlier. The validation dataset was concentrated on few class categories which was not even in training dataset. Later, we detect this from poor performance on models on validation dataset.

Tools

In the beginning, we have used Detectron2 but later changed to MMdetection framework for the project pipeline. This is because MMdetection provides more diverse options for backbones, models and augmentations by simply changing config files for training. Also, for inference, it provides much diverse modules including error analysis, fps benchmark calculation and confusion matrix. Finally, since we eager to use Weights&Biases for logging, MMdetection also supports logging using WandB (see Figure 7). We avoided vanilla Pytorch code because provided frameworks does optimized code for training and they already provided sufficient flexibility of settings we expected for the project.

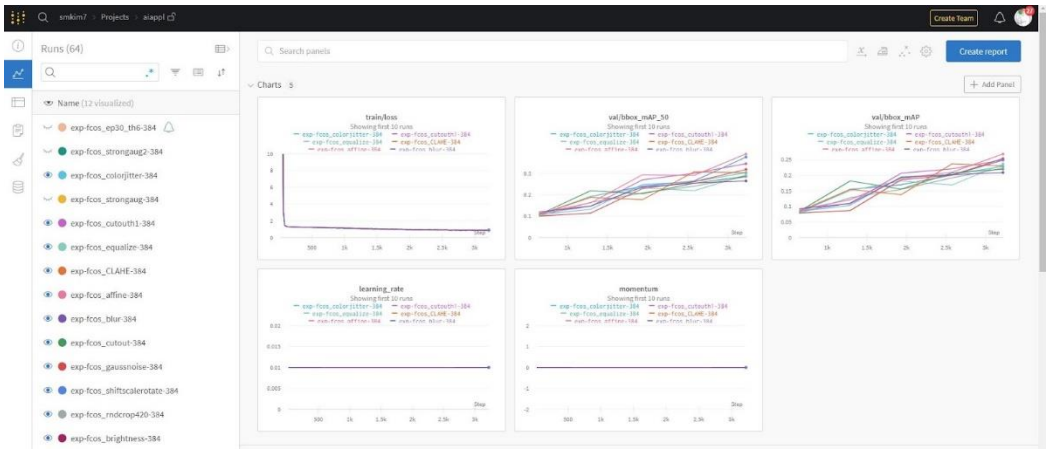


Figure 7 Weights & Biases Dashboard using WandB

For logging, we used WandB. We avoided using Tensorboard because WandB is much simpler and easier to visualize different metrics and results. We expected to gain from using hyperparameter search on image augmentations using WandB Sweep feature because there are so many combinations available for augmentations. However, we did not find a way to synchronize WandB Sweep feature with MMdetection training.

For image augmentations, rather than using default torchvision transforms modules, we used Albumentations library [2]. This is because from our experience, Albumentations is much faster in training and provide more diverse augmentation methods (see Table 1).

	Albumentations	imgaug	torchvision (Pillow backend)	torchvision (Pillow-SIMD backend)	Keras
RandomCrop64	**0.0017**	-	0.0182	0.0182	-
PadToSize512	**0.2413**	-	2.493	2.3682	-
HorizontalFlip	0.7765	2.2299	**0.3031**	0.3054	2.0508
VerticalFlip	**0.178**	0.3899	0.2326	0.2308	0.1799
Rotate	3.8538	4.0581	16.16	9.5011	50.8632
ShiftScaleRotate	**2.0605**	2.4478	18.5401	10.6062	47.0568
Brightness	**2.1018**	2.3607	4.6854	3.4814	9.9237
ShiftHSV	**10.3925**	14.2255	34.7778	27.0215	-
ShiftRGB	2.6159	**2.1989**	-	-	3.0598
Gamma	1.4832	-	**1.1397**	1.1447	-
Grayscale	**1.2048**	5.3895	1.6826	1.2721	-

Table 1 Times in seconds per image for different augmentation libraries [1]

Experiments

To optimize the model with our collected dataset, we have tried different experiment on variety of hyperparameters: optimizer, model, backbone, augmentation, and thresholds. To validate the model performance on different settings, we have selected few metrics that are widely used in object detection task. We used mAP50 and mAP for validation dataset. We did not highly rely on mAP75 (strict IoU option) because we don't think our manual labeling on bottles are 100% consistent and strict, so we used mAP50 as alternative option. Since it is easy to improve mAP merely using larger models and higher image resolution, we used fps, runtime, and memory as supplementary metrics to find out the speed and size of experimented models.

We have selected batch size of 4 for experiments and decreased it for large models if not fits within memory. Also, we have not used any learning rate scheduler, so learning rate is fixed during experiments phase. 384x384 image resolution and pretrained models are used. Moreover, Horizonal RandomFlip with 50% probability is included for training dataset in default. Pretrained models were all available from MMdetection github repository [3]. Note that the experimental results are different from Progress Review Meeting 3 because we have updated both training and validation dataset after we've found the error on splitting thus the performance is much higher than before.

Before we take experiments on different hyperparameters we have made five hypothesis that seems intuitive sense.

1. Increasing the batch size will help since the update of model parameters are updated on larger batch. Therefore, our final model uses larger batch size than the experiment phase.
2. RandomFlip must help model training. This is because real life images on whiskey bottles include several horizonal flipped images. However, we should avoid VerticalFlip. Therefore, we by default included RandomFlip augmentation with 50% probability.
3. TTA – Test Time Augmentation – must help performance on validation dataset. This is widely used in different Kaggle competitions, so we decided to include them.
4. Strong image augmentation is essential for training whiskey dataset because it is easy to overfit on training dataset even with small models from previous experiments.
5. Pretrained models reduce the time for training. We directly brought pretrained models from MMdetection github repository. Most of cases pretrained models help training and experiments.

For architectures, we have selected not only the popular models used in Kaggle, but unpopular models because we cannot easily assume models performed well on public COCO dataset will also perform well on our whiskey dataset because the data distribution of two datasets are not equal. Memory and fps are brought directly from MMdetection github repository. Here, we can see general tradeoff between speed and accuracy (see Table 2). Cascade_RCNN gives best result on both mAP50 and mAP, but with lowest fps. Models based on transformer architectures including DETR and Deformable DETR performs badly, either diverging or fails to get over 0.01 mAP. This was also true for YOLO models. Both YOLOv3 and YOLOX also receives <0.01 mAP performance. We tried smaller learning rate and gradient clipping, but they still diverge. Later we found that we can get reasonable performance with YOLO models, but with much longer training and learning rate scheduling.

Architecture	mAP50	mAP	Runtime	fps	Memory(GB)
Cascade_RCNN	**0.931**	**0.75**	17m 53s	16.1	4.2
Faster_RCNN	0.854	0.641	**12m 27s**	21.4	3.8
Retinanet	0.873	0.689	**12m 33s**	18.6	**3.5**
DETR	<0.01	<0.01	-	-	-
Sparse_RCNN	0.165	0.12	-	-	-
FCOS	0.664	0.512	16m 31s	**22.7**	3.8

Table 2 Experiment results for different architectures

We selected two best architectures from above result, Cascade_RCNN and Retinanet, and tried with different backbone models. What was interesting was that Retinanet performance falls with larger backbone. We thought this is because of model overfitting to training dataset. In overall, Cascade_RCNN outperforms Retinanet even for larger backbones (see Table 3). However, we realized the gain of performance by using larger backbones like ResNest50 is not that large. For example, changing from ResNet50 to ResNeSt50 only increased mAP50 by 0.001, but triples the training time. This again could be because of overfitting, and this might be resolved using strong image augmentations.

Architecture	Backbone	mAP50	mAP	Runtime	fps	Memory(GB)
Cascade_RCNN	Resnet-50	0.931	0.75	17m 53s	16.1	4.2
	HRNet-18	0.924	0.754	30m 48s	11.0	7.0
	ResNeSt-50	**0.932**	**0.762**	50m 36s	-	-
Retinanet	Resnet-50	0.873	0.689	**12m 33s**	18.6	**3.5**
	PVTv2-b4	0.838	0.656	29m 4s	-	17.0

Table 3 Experiment results for different backbones

We tried three different optimizers that are most popular – SGD, Adam and AdamW – with three different learning rates. Interestingly, Adam and AdamW performed really bad with our settings and it easily diverges even with low learning rate (see Table 4). Therefore, we stick with SGD optimizer. Learning rate is not important from this experiment because we expect to use learning rate scheduler for our final models anyway.

Optimizer	Learning Rate	mAP50	mAP
SGD	0.01	**0.931**	**0.75**
	0.005	0.917	0.713
	0.001	0.79	0.646
Adam	0.01	-	-
	0.005	-	-
	0.001	0.516	0.401
AdamW	0.01	-	-
	0.001	-	-

Table 4 Experiment results for different optimizers and learning rates

We tried different image augmentations using Albumentations library. Here, StrongAug is a combination of four image augmentations we experimented here. Different from pervious experiments, we also logged metrics for training dataset to see overfitting/underfitting behaviour of models. Comparing results from None to other augmentations, we can imply applying single augmentations did not really provide impressive performance gap just with 5 epochs (see Table 5). On interesting implication is that even with 5 epochs, there is mAP performance drop on training dataset on strong augmentations. However, it performs well on validation dataset, with highest mAP50. The performance gap between training and validation is lowest on StrongAug, so we decided to use strong augmentations on future works.

Augmentations	validation		train	
	mAP50	mAP	mAP50	mAP
None	0.931	0.75	0.989	0.821
RandomBrightnessContrast	0.935	0.753	0.989	0.811
ShiftScaleRotate	0.93	0.749	0.985	0.777
GaussianNoise	0.93	**0.764**	0.989	**0.83**
StrongAug	**0.948**	0.741	0.985	0.772
Cutout (1 big hole)	0.926	0.744	0.989	0.816

Table 5 Experiment results for different image augmentations

Finally, we tested on different thresholds on validation configurations. There are several thresholds we can tune, but we concentrated on thresholds we can tune on test configs, iou and score. The purpose of this experiment is not to look at which thresholds work best because for example as we train for longer epochs, it will make sense to use higher score thresholds as the model will become more confident on its class predictions. Instead, we want to look how metrics behave with different thresholds. From our experiment, we see mAP is not sensitive on iou thresholds, but it is highly dependent to the score thresholds (see Table 6). In conclusion, we decided to take some time on tuning score thresholds on future models.

valid	thresholds	mAP50	mAP
score	0.05 (default)	0.931	0.75
	0.1	0.919	0.741
	0.2	0.905	0.731
	0.4	0.886	0.717
rpn_iou	0.7 (default)	0.931	0.75
	0.6	0.931	0.752
	0.8	0.931	0.753
rcnn_iou	0.5 (default)	0.931	0.75
	0.4	0.931	0.75
	0.6	0.931	0.75

Table 6 Experiment results for different thresholds

After experiments, for future models we have made slight changes for improvement. First, we increased batch size to 24 (for 384x384) or 16 (for 512x512). Also, we used new normalization mean and std values that are directly calculated from our whiskey training dataset. For scheduling, we used steplr that reduces halves learning rate each 10 epochs.

Error analysis

Now with in-depth experiments with different hyperparameters, we tried to improve models using error analysis. We used confusion matrix, classwise Average Precision, logging best and worst images and precision-recall curve of different evaluation metrics [4]. These analysis modules provide feedback on our model performance and provide meaningful direction to update our model.

Classwise Average Precision

It is for sure that detectors cannot perform equally well on different classes. For example, models can perform bad on classes with lacking number of datasets because our dataset is not equally balanced. Also, some categories have similar bottle shape like Glenfiddich 12 and Glenfiddich 18, created from same company. Therefore, after training it is helpful to log AP scores on different classes and realize which classes model suffers to detect boxes accurately. After each training, we logged classwise AP on both train and validation dataset (see Figure 8). We diagnose whether the dataset is labelled wrong on low AP classes.

category	AP	category	AP	category	AP
Aberfeldy 12	0.876	Aberlour 12	0.831	Ardbeg 10	0.811
Balvenie 12	0.844	Bowmore 12	0.796	Bushmills	0.827
Chivals Regal 12	0.868	Cragganmore 12	0.785	Famous grouse	0.789
Glen Grant 10	0.871	Glenfiddich 12	0.764	Glenfiddich 18	0.844
Glenlivet 12	0.800	Highland Park 12	0.853	J-B	0.816
Jim Beam	0.828	Johnnie walker black	0.819	Johnnie walker red	0.838
Laphroaig 10	0.822	Macallan 12	0.801	Maker-s Mark	0.861
Monkey shoulder	0.830	Nikka coffey	0.838	Singleton 12	0.873
Talisker 10	0.840	Wild Turkey 101	0.817	Woodford Reserve	0.880

category	AP	category	AP	category	AP
Aberfeldy 12	0.850	Aberlour 12	0.892	Ardbeg 10	0.793
Balvenie 12	0.760	Bowmore 12	0.882	Bushmills	0.647
Chivals Regal 12	0.899	Cragganmore 12	0.877	Famous grouse	0.766
Glen Grant 10	0.872	Glenfiddich 12	0.900	Glenfiddich 18	0.870
Glenlivet 12	0.793	Highland Park 12	0.955	J-B	0.850
Jim Beam	0.832	Johnnie walker black	0.881	Johnnie walker red	0.918
Laphroaig 10	0.777	Macallan 12	0.883	Maker-s Mark	0.867
Monkey shoulder	0.866	Nikka coffey	0.897	Singleton 12	0.849
Talisker 10	0.866	Wild Turkey 101	0.810	Woodford Reserve	0.842

Figure 8 Average Precision scores of each class. Above: Train dataset, Below: Validation dataset

Confusion Matrix

Confusion matrix was another helpful inference module for error analysis. We look at false negatives and false positives of bounding boxes in this case. Confusion matrix includes background class on both ground truth (row) and prediction (column). Therefore, it gives percentage of bounding boxes predicted on background (false positive) and undetected bounding boxes where it should be (false negative). In this figure, we can directly visualize models from right confusion matrix performs much better than left. This is evident from higher percentage on diagonal line and lower percentage on last column. This visualization was helpful to tune score thresholds and further classwise analysis (see Figure 9). Appendix A includes enlarged version of confusion matrices.

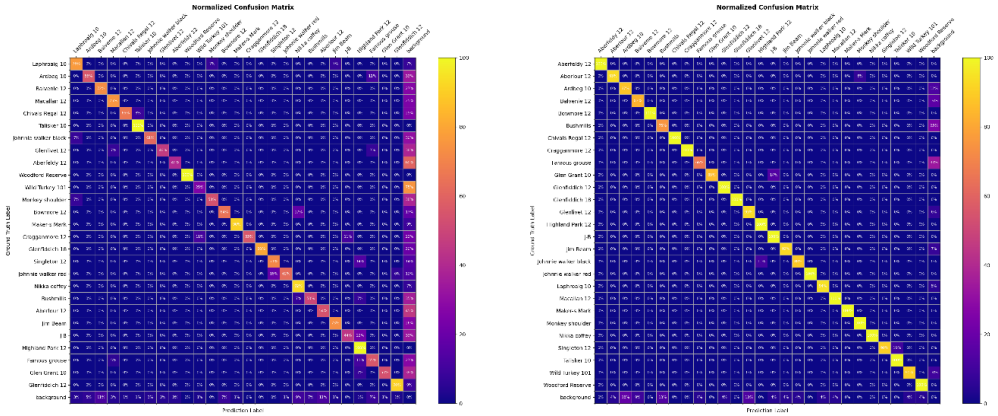


Figure 9 Confusion matrix of validation dataset. Left: Bad model, Right: Final model

Best/Worst examples

We also directly visualize example of best and worst images. We logged top 10 and bottom 10 images on validation dataset respect to mAP scores. Especially, worst examples were meaningful feedback because it explains in which cases model fails to detect well. For example, worst examples we've faced include bad resolution, multiple bottles and occlusion. The Images in Figure 10 are examples of bad images for models we trained in the beginning of the project. We can see it suffers from images with multiple bottles, enlarged bottle and bottle that is not from 27 categories in our dataset. This leads to model predicting several bounding boxes with low confidence scores.

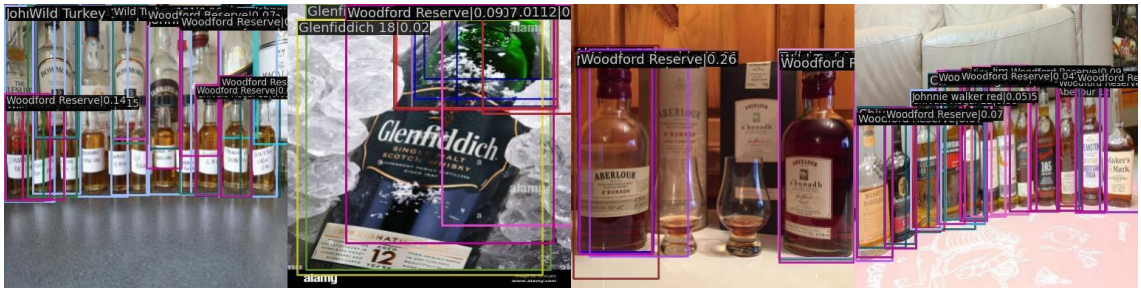


Figure 10 Worst 4 images on bad model

Figure 11 presents example of worst 4 images with our final models. We can see even the worst examples seem accurate predictions with high score confidence. As examples of failed cases (multiple bottles/enlarged bottle/unknown bottle) are not logged in this case, this implies our final model well predicts on cases that suffer from initial models.

Figure 11 Worst 4 images on final model

Precision-recall curves

We used series of precision-recall curve that is inspired by Diagnosing Error in Object Detectors by Derek Hoiem et al [4]. This module is provided by MMdetection so we can easily use them with our custom dataset. The plot shows different evaluation metrics. This is helpful to diagnose which error types are dominated from the model. There are several possible errors on object detection pipeline such as classification error, bounding box error and others. Also, we can plot curves on overall classes, each classes and different bounding box sizes, but we usually rely on plots averaged on all classes for the project.

Figure 12 presents precision-recall curves for bad(left) and good(right) models. For good model, we can easily see the error is dominated from localization error. This is because AP increases from 0.972 to 0.999 by changing IoU from 0.75(C75) to 0.5(C50). We suspect this is because of our manual labeling of bounding boxes are imperfect in our dataset. On the other hand, we see that the error of bad model is dominated by false positives since AP increases from 0.667 to 0.917 by removing error from false positives (Red area). This was very helpful in understanding which part our model fails and let us know the direction to improve.

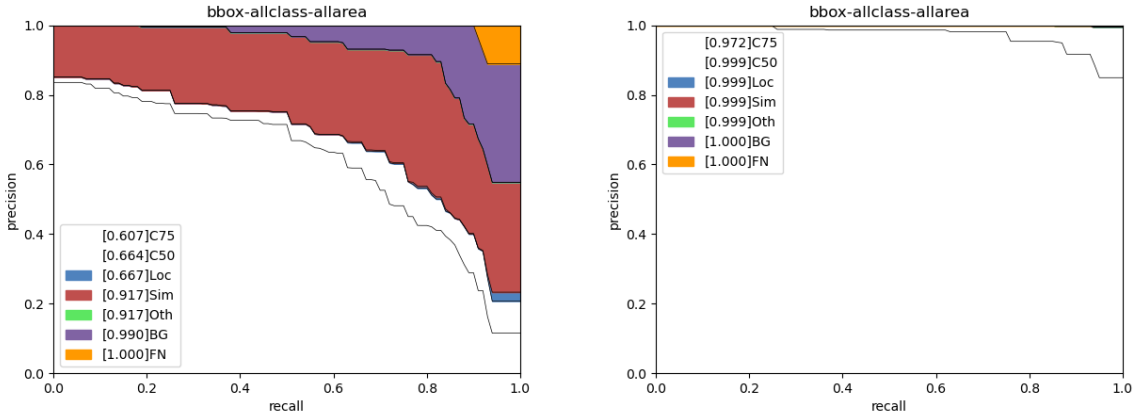


Figure 12 Error analysis precision-recall curves. Left: bad model. Right: final model

Final Implementation

Our final implementation are models with following specifications:

- Architecture/Backbone: Cascade-RCNN/ResNet50
- Epochs/Batch size: 30/24(384x384) and 16(512x512)
- Resolution: 384x384 and 512x512
- TTA: two image scales and no flip/horizontal flip. 4 TTAs in total
- Optimizer/Scheduler: 0.01 SGD/StepLr gamma=0.5 with step=[10, 20], no warmup
- Augmentations: RandomBrightnessContrast + ShiftScaleRotate + Cutout + GaussNoise with all 50% probability. Except for Cutout (which uses 1 large hole instead of default small 8 holes), all follows default parameters
- Thresholds: score threshold of 0.1, other thresholds keep as default

For training, we trained first 20 epochs with 384x384 resolution images and finetuned with 512x512 resolution images for final 10 epochs.

Our final model is inappropriate for real time detection because we benchmarked 6.3fps, which is much lower than our target 30 fps. So, we have decided to also provide YOLOX-s model which is trained for 120 epochs only with 384x384 resolution. Although the performance drops, it achieves over 0.5 mAP50 and benchmark 42fps, which is much faster than our target.

Web Application

A simple web application is implemented – Flask was used to serve the AI model and React.js was used to interact with users. The users will be able to get prediction of their images and download the labeled image (see figure 13). Details of the web application will be discussed in the final video presentation.

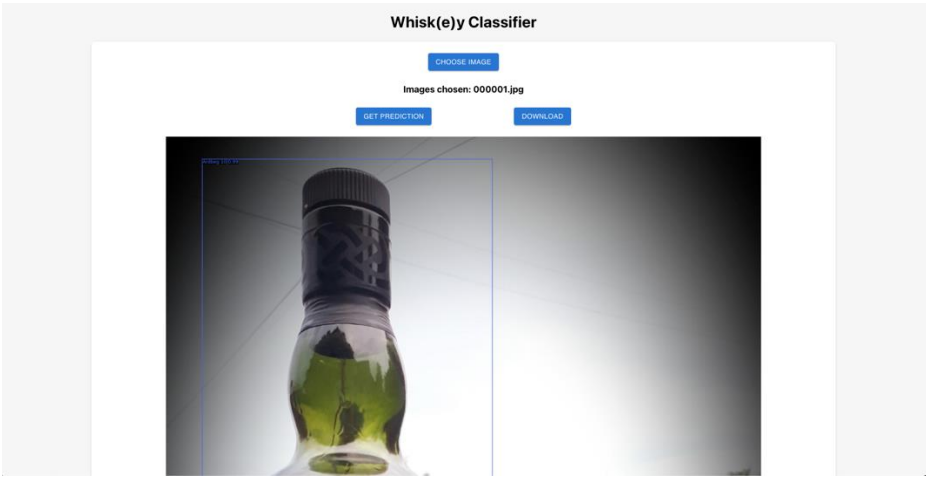


Figure 13 User interface implementation of whiskey classifier

Results

At the end, we tested on our final model with two different types - Cascade-RCNN both with and without Test Time Augmentation(TTA) and YOLOX-s only without TTA. Test datasets are not used anytime throughout our project so it well reflects unseen images. There is a performance gap between validation and test dataset, but the difference is not larger than we expected. Of course, it is possible to push up the performance using popular Kaggle techniques including pseudo-labeling or ensemble methods, but we believe the result is at satisfactory level.

Results		Test		Validation		Fps (Calculated)
Models	**TTA**	**mAP50**	**mAP**	**mAP50**	**mAP**	
Cascade_RCNN	4 TTAs	0.937	0.711	0.999	0.848	6.3
	None	0.921	0.685	0.950	0.785	21.9
YOLOX-s	None	0.558	0.337	0.590	0.374	42

Table 7 Results of the project

Difficulties

Our major source of difficulties comes from few assumptions we have made during the project which was wrong and critical for the project which leads to wasting time. First, the biggest mistake is that we assumed the Roboflow will split our datasets in stratified manner. We assumed out train and validation dataset will have similar class distribution. However, in fact it splits in biased way, so few classes are concentrated to validation dataset, which is not included in the training dataset. We suspect this later from the experiment that mAP plateaus at about 0.5 whatever models we use. Second, we assumed that MMdetection uses deterministic=True parameter as default when they set random seed. To create reproducible results, it is essential to fix seed and MMdetection provides api for this. However, later we realized the experiment results change every time even if we fix parameters. We explicitly set deterministic=True later in the api function. However, these mistakes can be realized relatively early using WandB because it can easily visualize parameters and metrics, which is limited if we used default TextLogger in MMdetection.

Other difficulties we face raised from as we rely on deep learning frameworks. Although the frameworks we used are optimized to code, it sometimes lacking flexibility compared to using vanilla Pytorch. For example, if confusion matrix is not plotted as we expected or even it outputs error, it was difficult to debug the error or find out what mistakes we have made throughout the project pipeline. This was resolved as we take full week on debugging potential errors in the code not only we created but also looking inside the framework code.

Moreover, although we heavily used different error analysis approaches to deep learning models, it was not sufficient to the level to XAI. We had an idea to apply attention mechanisms on images like Grad-CAM, but we believed this is not helpful because our models mostly predict bottles correctly and we suspect the attention must be on bottles anyway. Also, most of the feedback we learned from error analysis was by increasing number of datasets. For example, if some class particularly has low mAP, we can increase the images of that class. Also, if worst 4 images of model are from case of multiple bottle images, then we can increase images of that type. Increasing data of object detection is time consuming task so we did not have time to try this.

Finally, our difficulties come from the fact that we made the dataset ourselves. Although we have revised the dataset few times, we cannot assure the quality of our dataset. For examples, precision of bottles labeled on bounding boxes may differ because of manual labeling (see Figure 14). This was our primary reasons not to rely on mAP75 metric because the cause of localization error is not from the model but from our manual labeling. If there exists any previous model trained for bottles available in public, we can use transfer learning so avoid manual labeling, but we couldn't find one.

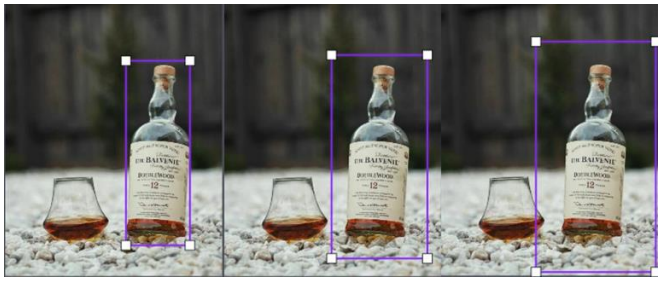


Figure 14 Different labeling precision

Limitations

The biggest limitations come from the fact that the performance and speed tradeoff of our created models. As the demo of our web application shows, the detection of whiskey bottles takes longer time than we expect and far from our primary objective to detect in real time. Therefore, we produced different versions of architectures, with lower performance but with faster speed.

Future Works

Possible future works include further improving the performance of the models using popular techniques used in Kaggle. This includes pseudolabeling, ensemble models or stochastic weight averaging which improves generalization of models thus may work better on unseen images. This is not conducted in the project because we believe our current result is already promising.

Other works may include expanding different whiskey bottle classes. Our project is somewhat impractical as the dataset only includes 27 categories, which is small subset of whiskeys that are sold around the world today. Also, this can be applied to different bottles that are not whiskeys. Our models can be used as transfer learning of other bottle types such as beer, fizzy drinks, or any other beverages of interest.

Furthermore, our web product could be more valuable if we included recommendation or information of whiskeys from input image. As our target audience of the project is anyone who is unfamiliar with whiskeys, recommending other whiskeys that share similar features detected from input image can be valuable. This extends to use another AI application of recommendation systems.

Conclusion

In conclusion, our project aims to build an application to help people who are not familiar with whiskey to understand whiskeys when they provide an image of whiskey. However, a complete application to provide detailed images would be hard to be implemented within limited time and resources. Therefore, we decided to work on the baseline of the application which is to build an AI model to classify whiskeys.

As the first step of the project, we have collected around ten thousand images from the internet using data crawler implemented by ourselves and those images were reviewed and around 3600 images were labelled. As a result, after excluding classes with small number of data, around 3400 images were used as the dataset of the project. While preparing the dataset, we have realized most of the images had white background, hence, we tried to collect images with various backgrounds – white and non-white, occlusion of the bottle, image including only part of the bottle enlarged, empty bottles without liquid inside and others.

In terms of the tools, PyTorch, WandB, MMdetection and Albumentations were used for building the AI model. With tools used, we achieved various kinds of experiments on architecture, backbone, learning rate, optimizers, image augmentations and thresholds of the model. With the results from the experiments, error analysis was done with four methods, and they are classwise average precision, confusion matrix, best/worst examples and precision-recall curves. As a result of experiments and error analysis, the final model was consisted of following

specifications:

- Architecture/Backbone: Cascade-RCNN/ResNet50
- Epochs/Batch size: 30/24(384x384) and 16(512x512)
- Image resolution: 384x384 and 512x512
- TTA: two image scales and no flip/horizontal flip. 4 TTAs
- Optimizer/Scheduler: 0.01 SGD/StepLr gamma=0.5 with step=[10, 20], no warmup
- Augmentations: RandomBrightnessContrast + ShiftScaleRotate + Cutout + GaussNoise with all 50% probability. Except for Cutout (which uses 1 large hole instead of default small 8 holes), all follows default parameters
- Thresholds: score threshold of 0.1, other thresholds keep as default

After the model was finalized, a simple web application was implemented to serve the model and interact with users using Flask and React.js.

References

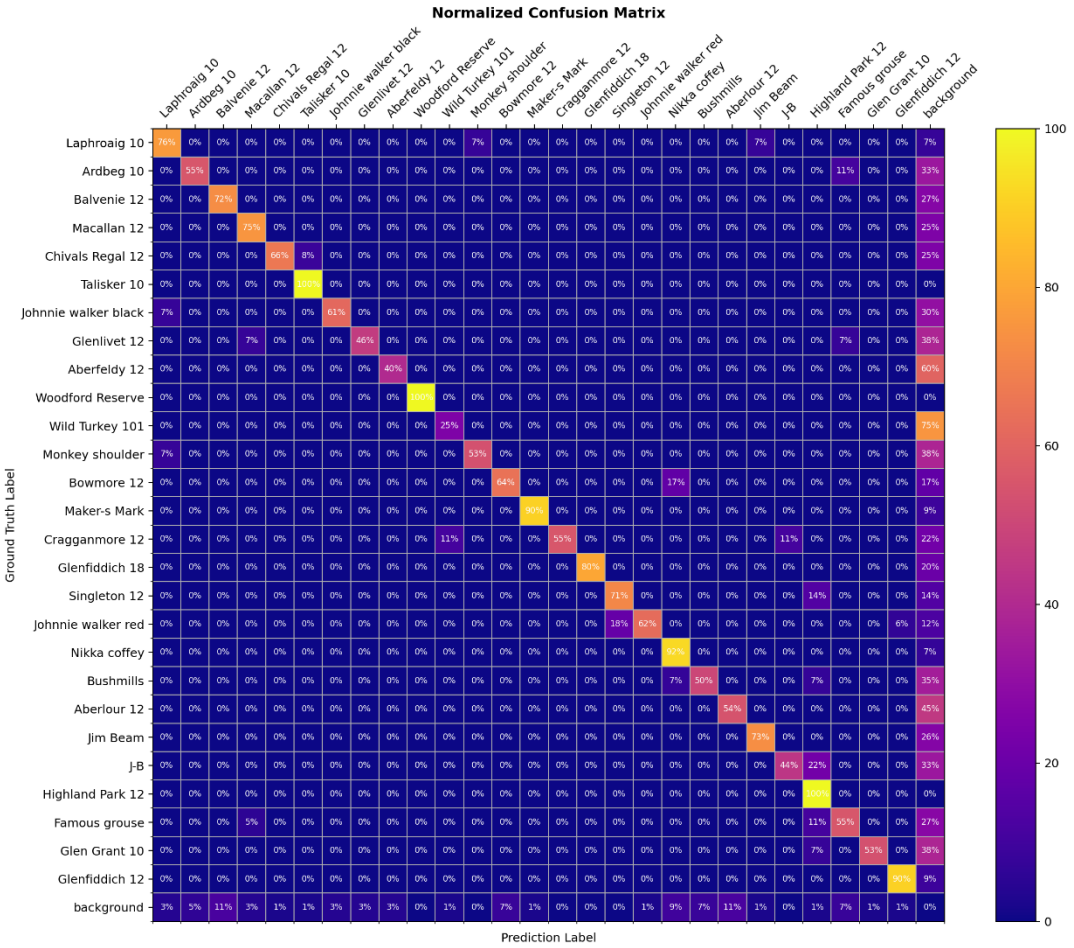
[1] https://arxiv.org/pdf/1809.06839.pdf

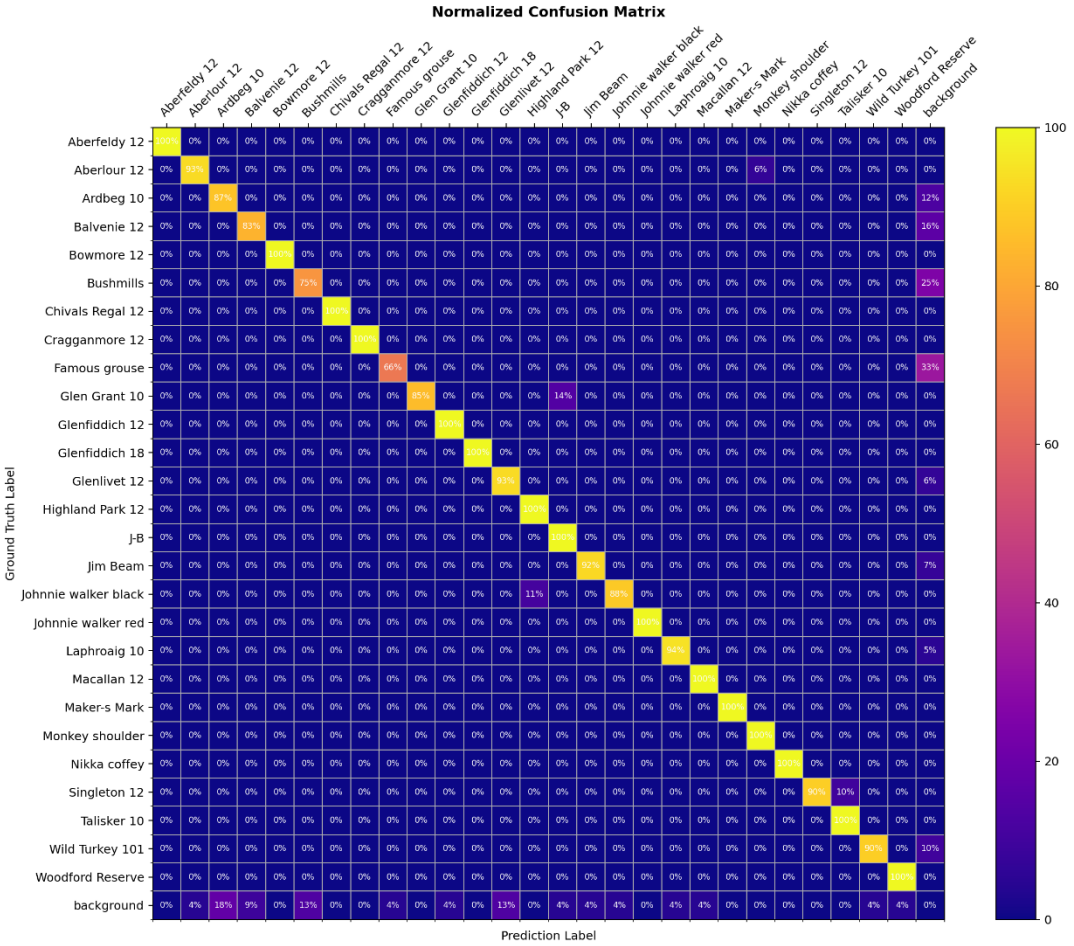
[2] https://albumentations.ai/

[3] https://github.com/open-mmlab/mmdetection

[4] http://dhoiem.cs.illinois.edu/projects/detectionAnalysis/

Appendix A: Enlarged Confusion matrix




Normalized Confusion Matrix
Aberfeldy 12
Aberlour 12
Ardbeg 10
Balvenie 12
Bowmore 12
Bushmills
Chivals Regal 12
Cragganmore 12
Famous grouse
Glen Grant 10
Glenfiddich 12
Glenfiddich 18
Glenlivet 12
Highland Park 12
J-B
Jim Beam
Johnnie walker black
Johnnie walker red
Laphroaig 10
Macallan 12
Maker-s Mark
Monkey shoulder
Nikka coffey
Singleton 12
Talisker 10
Wild Turkey 101
Woodford Reserve
background
Ground Truth Label
Prediction Label
100
80
60
40
20
0

Appendix B: WandB charts for training